Sound and Complete Reactive UAV Behavior using Constraint Programming

Hoang Tung Dinh
Mario Henrique Cruz Torres
Tom Holvoet

Report CW 707, September 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Sound and Complete Reactive UAV Behavior using Constraint Programming

Hoang Tung Dinh
Mario Henrique Cruz Torres
Tom Holvoet

Report CW 707, September 2017

Department of Computer Science, KU Leuven

Abstract

Realizing autonomous behavior of UAVs is a complex endeavor. The behavior needs to be goal-directed (1), while adhering to safety constraints (2), and while dealing with contingency situations (3). A purely imperative approach is unsuitable for this purpose, since such approach is unable to manage the sheer complexity of the behavior, leading to erroneous reactions to unforeseen situations in the environment, or situations that are simply not covered. In this paper, we present an innovative declarative approach to specify the autonomous behavior of UAVs. The approach combines a planning-based approach with constraint-programming for specifying safety and behavior constraints. The formal specification can be verified to be realizable. From the specification, we generate an execution policy, which can be proven to be sound and complete by construction if the specification is realizable. If the specification is not realizable, e.g., due to conflicting constraints, the generator indicates the source of the problem.

We illustrate our approach on a case study that uses a UAV for air quality monitoring and show how the approach assists the developer to specify correct autonomous behavior. The resulting behavior is deployed on an on-board computer of a UAV and tested on the field.

Keywords : Constraint Programming, Unmanned Aerial Vehicles, Autonomous Software.
CR Subject Classification : Computing methodologies, Artificial intelligence, Control methods

Sound and Complete Reactive UAV Behavior using Constraint Programming

Hoang Tung Dinh, Mario Henrique Cruz Torres, Tom Holvoet

***Abstract*— Realizing autonomous behavior of UAVs is a complex endeavor. The behavior needs to be goal-directed (1), while adhering to safety constraints (2), and while dealing with contingency situations (3). A purely imperative approach is unsuitable for this purpose, since such approach is unable to manage the sheer complexity of the behavior, leading to erroneous reactions to unforeseen situations in the environment, or situations that are simply not covered. In this paper, we present an innovative declarative approach to specify the autonomous behavior of UAVs. The approach combines a planning-based approach with constraint-programming for specifying safety and behavior constraints. The formal specification can be verified to be realizable. From the specification, we generate an execution policy, which can be proven to be sound and complete by construction if the specification is realizable. If the specification is not realizable, e.g., due to conflicting constraints, the generator indicates the source of the problem.**

We illustrate our approach on a case study that uses a UAV for air quality monitoring and show how the approach assists the developer to specify correct autonomous behavior. The resulting behavior is deployed on an on-board computer of a UAV and tested on the field.

I. Introduction

Recent advances in both hardware and software for UAVs open the door to several applications. As UAV missions become more and more complicated, the demand for autonomous behavior is increasing. However, defining the autonomous behavior for UAVs is complex. First, this behavior needs to be goal-directed, i.e., given a set of actions that a UAV can perform (such as take-off, navigate to a waypoint, hover, capture images, etc.), the system must come up with a sound plan to achieve the mission goals, taking into account the situation in the environment. Second, safety requirements must be adhered to. Examples are collision avoidance, respecting no-fly zones, only fly if the battery level is sufficient, only land in safe spots, etc. Complex as the combination of both already is, the environment in which a UAV operates is typically dynamic and often unpredictable. The UAV must account for contingencies and adapt its behavior accordingly, always adhering to the stipulated safety constraints. This reaction must be fast and correct in terms of mission goals and safety requirements.

To accomplish this, developers must be able to specify and verify the behavior of an autonomous UAV. At run-time, the autonomous behavior must be sound and complete. The behavior is sound if the UAV behaves correctly according to the given specification. The behavior is complete if there is no situation where there are no actions the UAV can take to satisfy the specification. If there exists no sound and complete behavior that can satisfy a specification, we call the specification *unrealizable*.

The authors are with imec-DistriNet, KU Leuven, 3001 Leuven, Belgium. Email: {hoangtung.dinh, mariohenrique.cruztorres, tom.holvoet}@cs.kuleuven.be

Defining the autonomous behavior of UAVs imperatively, by specifying *how* the UAV should behave (e.g., by constructing Finite State Machines [1], [2]) is hard, not manageable or maintainable, and not verifiable, due to the explosion of possible states to be taken into account. In this paper we propose an approach to specify and generate the autonomous behavior of UAVs. In this approach, the actions, goals and constraints are specified in a declarative way, i.e., by defining rules about *what* the expected behavior of the UAV should accomplish. Such specification represents the minimal requirements. The actual behavior, i.e., the actions that are selected in a particular situation, is generated based on the specification.

In concreto, the behavior of a UAV is specified by a set of logical rules. The rules take into account mission goals, the actions a UAV is able to perform and the safety requirements that the UAV must conform to. This specification combines concepts from a classical planning approach - such as states, actions, goals - with constraint programming - imposing additional limitations on acceptable behavior. The behavior specification is then automatically translated to a set of constraint satisfaction problems (CSPs) [3] which are solved off-line. The solutions of the CSPs form an *execution policy* that maps each possible state of the world, that concerns the behavior specification, to a set of actions to be executed, which represents the *reactive* behavior of the UAV. The generated policy is sound because it satisfies the behavior specification, i.e., all the specified rules, by construction. On the one hand, if the specification is realizable, all the CSPs are feasible and the achieved policy is complete. On the other hand, if the specification is unrealizable, some CSPs are infeasible which indicates the situations leading to the unrealizability of the behavior specification.

The generated policy is used at runtime to achieve the autonomous behavior of the UAV. Whenever the state of the world changes, the generated policy decides upon which actions to execute.

This paper brings two contributions to the autonomous vehicle research:

- We present a new formal way to specify the behavior of a UAV and its mission
- We present a technique to generate sound and complete execution policies from our behavior specifications.

This paper is organized as follows. Section II discusses

related work. Section III presents our approach to formally specify the behavior of a UAV. Section IV details our approach to generate an execution policy from the behavior specification. Section V describes a case study where the proposed behavior specification and generation approach is applied to an air quality monitoring mission. Finally, Section VI draws conclusions and details possible future work.

II. Related work

Traditionally, robot behaviors are constructed manually by specifying *how* a robot should behave. Finite State Machines (FSMs) [1], [2] are the most commonly used model to represent robot behaviors. A well-known limitation of FSMs is that the number of states and transitions of a FSM exponentially grows as the complexity of the behavior arises. Recently, Behavior Trees (BTs) [4], [5] received attention in the robotics community as an alternative to FSMs thanks to their modularity. However, different from FSMs, Behavior Trees are not event-driven, which may not be suitable for safety critical behaviors.

Hand-coding robot behaviors is hard and one has to rely on simulation to check if the constructed behavior meets the specified requirements. To address this problem, approaches to automatically generate robot behaviors from formal mathematical-based representations of requirements have been proposed.

Doherty et al. [6] use Temporal Action Logic (TAL) [7] as a specification language. Several planners have been developed to generate plans from specifications written in TAL [8], [9]. However, those planners assume that the environment is fully controllable and do not take into account contingencies. At runtime, if failures are detected during the plan execution, replanning is triggered [10]. This approach is similar to the use of the Planning Domain Definition Language (PDDL) [11] in the planning community, where generic planners are developed to solve planning problems described in PDDL. Since these planning approaches rely on replanning at runtime to deal with contingencies, they do not guarantee the completeness of the behavior. The replanning may fail due to an unrealizable specification and this failure can only be detected at runtime. Besides that, replanning can be computationally expensive and take too long to execute at runtime.

Linear Temporal Logic (LTL) [12] is another specification language often used in robotics. There exists synthesis techniques to generate FSMs [13], [14] or BTs [15] from specifications written in some fragments of LTL. Using these synthesis approaches, the behavior is guaranteed to be sound and complete since all contingencies are taken into account in the generated behavior. The main limitation of the LTL approaches is their computational complexity.

III. Behavior specification

In this section we present a new approach for specifying the behavior of an autonomous UAV in a mission. In our approach, a *behavior specification* is a formal representation of all the concerns relevant to the UAV's behavior in the mission. The specification aggregates information about how the UAV perceives the world, the actions it can perform, the safety aspects of a mission and its goals. Fig. 1 shows how the behavior specification fits in the whole system, being used off-line to generate an *execution policy* that maps each possible state of the world to a set of actions to execute. The execution policy is used at runtime by the UAV, so that it knows which actions have to be executed in the occurrence of any changes in the environment.

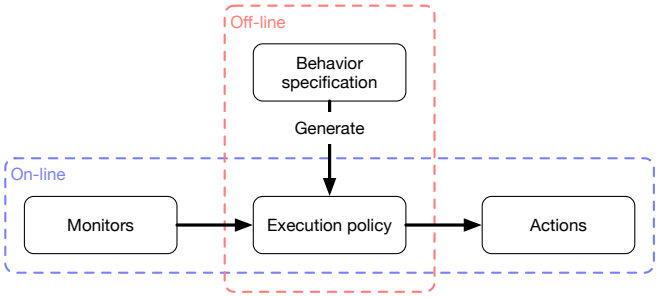

Fig. 1. From a behavior specification, an execution policy is automatically generated off-line. At runtime, the execution policy maps the state vector value given by the monitors to a set of actions to execute.

A behavior specification consists of four elements: (1) *state vector*, (2) *actions*, (3) *reaction rules* and (4) *goals*.

The *state vector* abstracts the world representation. It contains state variables about the environment and the system itself, which one is interested in.

An *action* represents a primitive behavior that the UAV can perform, e.g., take-off. An action is durative, i.e., it takes time. The specification of an action includes the preconditions for the activation and execution of the action, the desired effects the action will have and the controlled resources that the action requires exclusive control.

The *reaction rules* specify the reaction that the UAV has to guarantee during a mission.

The final element, *goals* represent the desired goals of the UAV's mission.

Each one of the above-mentioned elements is further explained in the following sub-sections.

A. State vector

A state vector $\mathbf{S}$ represents which states of the world and the UAV-system that the UAV is concerned with. Formally, a state vector is a set of measurable *discrete state variables* $\{S_0, S_1, ..., S_n\}$. Each state variable S_i represents a state that the UAV can measure. At runtime, the value of a state variable is updated by a *monitor*, which measures a specific aspect of the world or of the system. A monitor translates a measurement to a particular discrete value in a state variable.

It is the responsibility of developers to decide which aspects of the world and the UAV-system should be represented in the state vector and how they are discretized and monitored. E.g., to represent whether the battery level is below 5%, one can create the state variable $S_{battery} = \{bellow\_5, between\_5\_100\}$ and a monitor that constantly measures the battery level and updates the state variable to

the corresponding value at runtime. A state variable could also represent aspects related to history. E.g., one can define the state variable $S_{a\_executed} = \{true, false\}$ to represent whether action $\mathbf{a}$ has ever been executed and finished.

B. Actions

In our approach, the behavior of autonomous UAVs is about deciding which *actions* to execute based on runtime feedback information. Given a set of actions $\mathbf{A} = \{\mathbf{a_0}, \mathbf{a_1}, ..., \mathbf{a_n}\}$, the semantic meaning of each action $\mathbf{a}$ is defined by three properties: *preconditions*, *desired effects* and *controlled resources*. An action's preconditions and desired effects define when the action can be executed and what is the expected state after such action's execution. We also formally specify the controlled resources, so that it is possible to guarantee that there are no conflicts in the system regarding multiple actions trying to control the same resources simultaneously.

Preconditions: $\text{Pre}(\mathbf{a}, \mathbf{S})$ are predicates on the state vector $\mathbf{S}$ that must hold on the activation and during the execution of the action $\mathbf{a}$. An action can only be started if its preconditions hold. During the execution, if the preconditions of an action change from "hold" to "not hold", the action will be stopped.

Desired effects: $\text{Eff}(\mathbf{a}, \mathbf{S})$ are predicates on the state vector $\mathbf{S}$ that will hold if the action $\mathbf{a}$ is executed successfully. Currently, our approach only allows desired effects to be represented as equality and conjunctive predicates. An action can still be executed after its desired effects have been achieved, as long as its preconditions still hold. If the desired effects have been achieved and the action is still executing, the desired effects will be maintained.

Controlled resources: $\text{R}(\mathbf{a})$ is a set of resources over which the action $\mathbf{a}$ requires exclusive control during its execution. If two actions have at least one controlled resource in common, they cannot be executed in parallel. Examples of controlled resources are the rotors, or an air quality sensor. We currently assume that an action has at most one controlled resource. This assumption may be relaxed in future studies.

The preconditions of an action are a set of necessary requirements for the action to achieve its desired effects. If the action cannot achieve the desired effects, we call the action *infeasible*. In that case, the observable condition for an action to be infeasible is represented by a predicate $\text{Infeasible}(\mathbf{a}, \mathbf{S})$ and the predicate $\neg\text{Infeasible}(\mathbf{a}, \mathbf{S})$ is added to the preconditions of the action. By doing so, we guarantee that an infeasible action will not be activated or if the action is already executing, it will be stopped. E.g., action $\mathbf{navigate\_to\_A}$ is infeasible if there exists no path from the current position of the UAV to point A, i.e., $S_{path\_to\_A} = not\_exist$. Therefore, the predicate $\neg(S_{path\_to\_A} = not\_exist)$ is included in the preconditions of the action. Note that, it is up to the developer to decide how $S_{path\_to\_A}$ should be monitored (e.g., by executing a path planner at a fixed frequency).

Given the assumption above, a developer must model all possible infeasible conditions of actions in their preconditions. If a contingency event leading to the infeasibility of an action is not modeled, the behavior of the UAV is not guaranteed to be correct. However, it is possible to define a high-level infeasible condition that can capture different unknown contingency events. E.g., an action that navigates the UAV to a checkpoint can be considered as infeasible if the UAV cannot reach the checkpoint in 5 minutes after the activation of the action or when a path planner cannot find a feasible path from the current position of the UAV to the checkpoint.

Compositions of actions can be defined by logical rules to constrain or enforce the parallel execution of actions. Logical rules about action compositions involve the predicate $\text{Exec}(\mathbf{a})$ which represents whether an action is executing. E.g., the rule in Eq. (1) requires action $\mathbf{a_0}$ and $\mathbf{a_1}$ to always be executing in parallel. Note that, actions executed in parallel must have no conflict in their preconditions and desired effects as well as no controlled resource in common. If there is a conflict, the behavior specification is unrealizable and the conflict situation will be detected during the policy generation (Section IV).

$$\text{Exec}(\mathbf{a_0}) \Leftrightarrow \text{Exec}(\mathbf{a_1}) \tag{1}$$

C. Reaction rules

Safety requirements related to the reaction of a UAV are represented as logical rules on the state vector and the executed actions as in Eq. (2). The predicate $\text{Cond}(\mathbf{S})$ represents whether a logical condition holds on the state vector. The rule in Eq. (2) states that if a condition on the state vector holds, action $\mathbf{a}$ must be executing.

$$\text{Cond}(\mathbf{S}) \Rightarrow \text{Exec}(\mathbf{a}) \tag{2}$$

This rule allows developers to specify the UAV's reactions to contingency events in a declarative way. E.g., one can require the UAV to go home or land if its battery level is low. At runtime, the UAV is guaranteed to handle contingency events correctly according to the reaction rules. Conflicts among reaction rules will be reported during the policy generation so that the developer can fine-tune the specification.

D. Goals

We represent the goals of the UAV's mission as rules concerning the values on the state vector $\mathbf{S^G}$, i.e., the state vector value after the UAV completed its mission. There are two typical types of goal rules.

$$\text{Goal}(\mathbf{S^G}) \tag{3}$$

$$\text{Cond}(\mathbf{S}) \Rightarrow \text{Goal}(\mathbf{S^G}) \tag{4}$$

The first type of goal rules (Eq. (3)) states that a condition must hold at the end of the mission. E.g., one may want the UAV to always be landed at the end of the mission. The second type of goal rules (Eq. (4)) represents *conditional goals* which only need to be achieved if the goals are feasible. The condition for a goal to be feasible is represented by the predicate $\text{Cond}(\mathbf{S})$ in Eq. (4). E.g., one may want the UAV to visit a checkpoint only if there exists a feasible path from the current position of the UAV to the checkpoint. At

runtime, the UAV continually assesses the feasibility of the conditional goals to decide whether to pursue them or not.

Given the state vector, actions, reaction rules and goal rules, we generate an execution policy $\pi(\mathbf{S}) = \{\mathbf{a_i}\}$ mapping each state vector value to a set of actions to execute (Section IV). At runtime, whenever there is a change to a state variable in the state vector, the UAV looks up at the generated policy to select the actions to execute (as shown in Fig. 1).

E. Example

We present a simple example of a behavior specification. Assuming that a UAV needs to fly to a checkpoint and land there. However, we only allow the UAV to fly if the battery level is above a predefined threshold. If the battery level of the UAV is below the threshold, the UAV must stop flying and land immediately. We define the state vector as follows.

$$\begin{aligned} \mathbf{S} &= \{S_{flying}, S_{dest}, S_{battery}\} \\ S_{flying} &= \{flying, landed\} \\ S_{dest} &= \{not\_reached, reached\} \\ S_{battery} &= \{below, above\} \end{aligned} \tag{5}$$

The following actions are defined for the mission.

- **TakeOff**:
  - Preconditions: $S_{flying} = landed$
  - Desired effects: $S_{flying} = flying$
  - Controlled resource: *rotors*
- **NavigateToPoint**:
  - Preconditions: $S_{flying} = flying$
  - Desired effects: $S_{dest} = reached$
  - Controlled resource: *rotors*
- **Land**:
  - Preconditions: $S_{flying} = flying$
  - Desired effects: $S_{flying} = landed$
  - Controlled resource: *rotors*

The following reaction rule states that if the battery level of the UAV is below a threshold, the UAV must land.

$$(S_{flying} = flying) \wedge (S_{battery} = below) \Rightarrow \text{Exec}(\mathbf{Land}) \tag{6}$$

The goals are specified in Eq. (7) and (8). The rule in Eq. (7) states that if the battery level is above the threshold, the UAV must try to reach the destination. The rule in Eq. (8) requires the UAV to land at the end of the mission.

$$S_{battery} = above \Rightarrow S^G_{dest} = reached \tag{7}$$

TABLE I

A SOUND AND COMPLETE EXECUTION POLICY

S_{flying}	S_{dest}	$S_{battery}$	Actions
landed	*not_reached*	*above*	**TakeOff**
landed	*not_reached*	*below*	**no-op**
flying	*not_reached*	*above*	**NavigateToPoint**
flying	*not_reached*	*below*	**Land**
flying	*reached*	*above*	**Land**
flying	*reached*	*below*	**Land**
landed	*reached*	*above*	**no-op**
landed	*reached*	*below*	**no-op**

$$S^G_{flying} = landed \tag{8}$$

Table I shows a sound and complete execution policy generated from the specification. The notation **no-op** indicates that no action needs to be executed.

IV. EXECUTION POLICY GENERATION USING CONSTRAINT PROGRAMMING

In this section we present how to generate an execution policy from a behavior specification described in the previous section. The process of generating an execution policy from a behavior specification is illustrated in Fig. 2. Recall that an execution policy is a function mapping each state vector value to a set of actions. We calculate the mapping result of a state vector value by solving a classical planning problem. The solution of the planning problem is a sequence of *execution steps* that transform the given state vector value to a state vector value that satisfies all the goal rules (see Section III-D). Each execution step is a set of actions to be executed in parallel. The first execution step of the solution is used as the mapping result from the given state vector value. The complete execution policy is generated by calculating the mapping results of all possible state vector values.

We model each classical planning problem mentioned above as a constraint satisfaction problem (CSP). In [16], Bartak et al. summarize several constraint satisfaction models for sequential planning problems, i.e., problems where only one action is allowed to execute at the same time. In this paper, we extend the constraint model described in [16] to support multiple actions to be executed in parallel.

Given a state vector value $\mathbf{S^0}$, we find a plan of length n, corresponding to n execution steps, transforming $\mathbf{S^0}$ to the goal state vector $\mathbf{S^n}$ which satisfies all the goal rules. To find the plan with the shortest length, we start solving the CSP with $n = 1$ and then gradually increase n by 1, until a feasible plan is found or a maximal value of n is reached.

Fig. 3 shows the decision variables of a CSP. Since each action has either one or no controlled resource and actions with the same controlled resource cannot be executed together (see Section III-B), maximally $k = k_0 + k_1$ actions

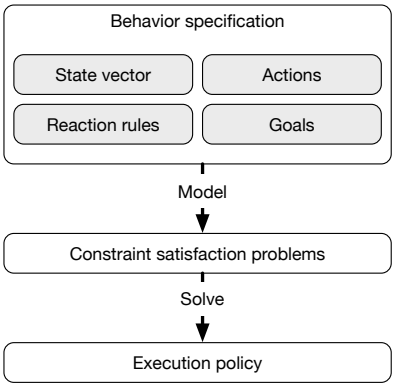

Fig. 2. To generate an execution policy, a set of constraint satisfaction problems (CSPs) are constructed from the behavior specification. Each CSP models a mapping result of a state vector value. All the CSPs are solved off-line to achieve the complete execution policy.

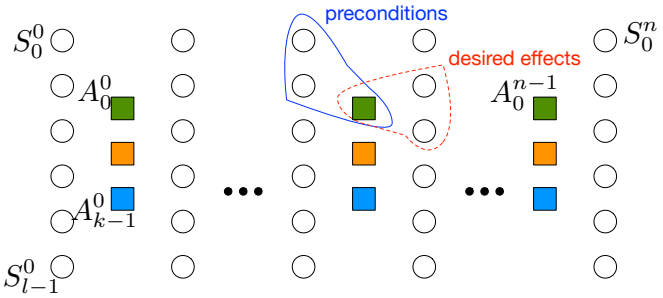

Fig. 3. Decision variables in a planning problem modeled as a CSP. The preconditions and desired effects of an action (represented by a square) show the connection between two instances of the state vector (represented by circles). This figure is partly based on Figure 1 in [16].

can be executed in parallel , where k_0 is the total number of controlled resources and k_1 is the number of actions having no controlled resource. Therefore, we represent a plan by $k \times n$ action variables with k action variables at each execution step. Since the plan has length n, there are $n+1$ state vectors. Each state vector is represented by a set of state variables.

It is possible to define the domain $\mathbf{D}$ of each action variable as all the specified actions in $\mathbf{A}$ and add constraints stating that two actions having the same control resource cannot be in the same execution step. However, doing so leads to a big model which is computationally expensive to solve. We therefore reduce the computational complexity of the constraint model by narrowing down the domain of each action variable as follows.

For the first k_0 rows of action variables (different rows of action variables are represented by different colors in Fig. 3), the domain of action variables in each row consists of only actions having the same controlled resource. For the next k_1 rows of action variables, the domain of action variables in each row consists of a single action having no controlled resource. In other words, each row of action variables corresponds to one controlled resource or a single action with no controlled resource. Since it could be the case that not all execution steps have exactly k actions to execute in parallel, we define a *null* action with empty preconditions and desired effects to fill in the empty position at each execution step. The *null* action is added to the domain of each action variable. This way of modeling reduces the domains of the action variables while ensuring that actions controlling the same resource are not executed in parallel since they are represented by a single action variable at each execution step.

For each action variable A_r^i connecting the state vector $\mathbf{S^i}$ and $\mathbf{S^{i+1}}$, the following constraints are added to impose the preconditions and desired effects [16]. The constraints are repeated for each action $\mathbf{a}$ in the domain of A_r^i.

$$\begin{aligned} A_r^i = \mathbf{a} &\Rightarrow \text{Pre}(\mathbf{a}, \mathbf{S^i}) \\ A_r^i = \mathbf{a} &\Rightarrow \text{Eff}(\mathbf{a}, \mathbf{S^{i+1}}) \end{aligned} \tag{9}$$

We assume that at each plan step, state variables which are not in the desired effects of A^i stay the same. This assumption, which is called frame axioms in the literature [16], is necessary to formulate a classical planning problem. This assumption does not affect the validity of the resulting policy since we create a plan for each possible value of the state vector. At runtime, every time the value of the state vector changes, the actions corresponding to the plan of the new state vector value will be executed.

Since in [16], at each execution step only one action is executed, the frame axioms can simply be defined as follows.

$$A_r^i = \mathbf{a} \Rightarrow S_j^i = S_j^{i+1}, \forall S_j \notin \text{Eff}(\mathbf{a}, \mathbf{S}) \tag{10}$$

However in our problem, k actions are executed in parallel at each step. Therefore, to represent the frame axioms, for each state vector S^i, we define k boolean vectors $\mathbf{B}^r, r \in \langle 0, k-1 \rangle$ having the same length v as the state vector. Each boolean vector $\mathbf{B}^r$ represents which state variables in the state vector $\mathbf{S^{i+1}}$ action A_r^i has effect on. Then, the frame axioms are defined as follows.

$$\begin{aligned} A_r^i = \mathbf{a} &\Rightarrow \mathbf{B}^r = \mathbf{B}(\mathbf{a}) \\ \Big(\bigvee_r B_j^r\Big) = \mathit{false} &\Rightarrow S_j^i = S_j^{i+1} \end{aligned} \tag{11}$$

where $\mathbf{B}(\mathbf{a})$ is a precomputed boolean vector. An element in $\mathbf{B}(\mathbf{a})$ is true if $\mathbf{a}$ has effect and is false if $\mathbf{a}$ does not have effect on the corresponding state variable. Eq. (11) states that a state variable must stay unchanged after an execution step if it does not belong to the desired effect of any action executed in that step.

Reaction rules (Section III-C) are added to the state and action variables at each execution step. The predicate $\text{Exec}(\mathbf{a})$ is translated to the predicate $A_r^i = \mathbf{a}$ in the CSP, where r is the row where the action variable domain containing $\mathbf{a}$. E.g., the rule in Eq. (2) is added to the CSP as follows.

$$\bigwedge_i \big(\text{Cond}(S^i) \Rightarrow A_r^i = \mathbf{a}\big) \tag{12}$$

Rules representing goals (Section III-D) are applied on the first state vector $\mathbf{S^0}$ and the final state vector $\mathbf{S^n}$. The rules in Eq. (3) and (4) are translated to the following constraints.

$$\text{Goal}(\mathbf{S^n}) \tag{13}$$

$$\text{Cond}(\mathbf{S^0}) \Rightarrow \text{Goal}(\mathbf{S^n}) \tag{14}$$

The resulting execution policy generated by solving all the CSPs is sound by construction, i.e., the policy is correct according to the behavior specification. It is because the policy satisfies all the constraints translated from the reaction rules and goal rules. The generated policy is complete if there exists a mapping result for each state vector value, i.e., all CSPs are feasible. The unrealizability of the specification can be detected when a CSP is infeasible. In that case, the corresponding state vector value will be reported so that one can refine the specification. Examples of an unrealizable behavior specification will be presented in Section V-D.

Note that, while the generated execution policy is sound and complete, it is not guaranteed to be optimal. It is because this policy generation approach only aims at finding a feasible policy that can satisfy the behavior specification

and does not take into account any optimality criterion. Our future work will focus on providing some guarantees on the optimality of the generated policy.

V. Case study - Air quality monitoring

We present a case study where our behavior specification and generation approach is applied to a mission taken from the SafeDroneWare[1] project. In the mission, the UAV has to monitor the air quality at three checkpoints A, B and C (can be in arbitrary order). The UAV takes off at $home$ location. To monitor the air quality at a checkpoint, the UAV needs to fly to the checkpoint and activate its air quality sensor. The UAV must hover while the air quality sensor is collecting data. If the UAV cannot reach a checkpoint (e.g., because the path planner cannot find a feasible path), it can ignore that checkpoint. After finishing taking samples at all possible checkpoints, the UAV should return to $home$. However, if the battery level of the UAV is below 20%, the UAV should abort the mission and return to $home$. In addition, if the battery level is below 5%, the UAV needs to land immediately, no matter whether it reached $home$ or not. The UAV also needs to support manual control mode if the boolean variable $manualcontrol$ is set to $true$.

In our notation in the following subsections, state variables and actions represented with a parameter, like $S_{sample}(x), x \in \{A, B, C\}$, represent multiple occurrences of such entities. E.g., the above state variable notation is a short version of three state variables S_{sample_A}, S_{sample_B} and S_{sample_C}.

A. State vector

We define the following state variables for the mission with their corresponding meanings.

$$\begin{aligned}
&\mathbf{S} = \{S_{flying}, S_{location}, S_{battery}, S_{sample}(x), S_{path}(y), \\
&\quad S_{manualcontrol}\},\ x \in \{A, B, C\},\ y \in \{A, B, C, home\} \\
&S_{flying} = \{flying, landed\} \\
&S_{location} = \{home, A, B, C, other\} \\
&S_{battery} = \{above_20\%, from_5_to_20\%, below_5\%\} \\
&S_{sample}(x) = \{not_collected, collected\},\ x \in \{A, B, C\} \\
&S_{path}(x) = \{not_exist, exist\},\ x \in \{A, B, C, home\} \\
&S_{manualcontrol} = \{false, true\}
\end{aligned} \tag{15}$$

B. Actions

We define the following actions for the mission.

- **TakeOff**
  - Preconditions: $S_{flying} = landed$
  - Desired effects: $S_{flying} = flying$
  - Controlled resources: *rotors*
- **Land**
  - Preconditions: $S_{flying} = flying$
  - Desired effects: $S_{flying} = landed$
  - Controlled resources: *rotors*
- $\mathbf{Navigate}(x), x \in \{home, A, B, C\}$
  - Preconditions: $S_{flying} = flying$, $S_{path}(x) = exist$
  - Desired effects: $S_{location} = x$
  - Controlled resources: *rotors*
- $\mathbf{Hover}(x), x \in \{A, B, C\}$
  - Preconditions: $S_{flying} = flying$, $S_{location} = x$
  - Desired effects: $S_{flying} = flying$, $S_{location} = x$
  - Controlled resources: *rotors*
- $\mathbf{CollectSample}(x), x \in \{A, B, C\}$
  - Preconditions: $S_{flying} = flying$, $S_{location} = x$, $S_{sample}(x) = not\_collected$
  - Desired effects: $S_{sample}(x) = collected$
  - Controlled resources: *air quality sensor*
- **ManuallyFly**
  - Preconditions: $S_{flying} = flying$, $S_{manualcontrol} = true$
  - Desired effects: $S_{manualcontrol} = false$
  - Controlled resources: *rotors*

The following composition action rule is added to enforce the UAV to hover while collecting air samples.

$$\text{Exec}(\mathbf{CollectSample}(x)) \Rightarrow \text{Exec}(\mathbf{Hover}(x)) \tag{16}$$

C. Reaction rules and goals

We now translate the description of the mission to reaction rules and goals. Note that, the current specification is unrealizable. We will show how the unrealizability can be detected during the generation of the execution policy.

The first rule states that whenever the user wants to take control of the UAV and the UAV is flying, the action **ManuallyFly** must be executed.

$$\begin{aligned}
&(S_{manualcontrol} = true) \wedge (S_{flying} = flying) \Rightarrow \\
&\text{Exec}(\mathbf{ManuallyFly})
\end{aligned} \tag{17}$$

The second rule states that if the battery level is between 5% and 20%, the UAV must navigate to $home$.

$$\begin{aligned}
&(S_{battery} = from_5\%_to_20\%) \wedge (S_flying = flying) \wedge \\
&\neg(S_location = home) \Rightarrow \text{Exec}(\mathbf{Navigate}(home))
\end{aligned} \tag{18}$$

The last rule requires the UAV to land if the battery level is below 5%.

$$\begin{aligned}
&(S_{battery} = below_5\%) \wedge (S_flying = flying) \Rightarrow \\
&\text{Exec}(\mathbf{Land})
\end{aligned} \tag{19}$$

The goals of the mission are to collect air samples at A, B and C if there exists feasible paths to those checkpoints and then return to home and land. The goals are represented by the three following rules.

$$(S_{path}(x) = exist) \Rightarrow S^G_{sample}(x) = collected \tag{20}$$

$$S^G_{location} = home \tag{21}$$

$$S^G_{flying} = landed \tag{22}$$

[1] https://www.imec-int.com/en/what-we-offer/research-portfolio/safedroneware

$S_{flying} = flying$	$S_{path}(A) = exist$
$S_{location} = other$	$S_{path}(B) = exist$
$S_{battery} = below\_5\%$	$S_{path}(C) = exist$
$S_{sample}(A) = collected$	$S_{path}(home) = exist$
$S_{sample}(B) = collected$	$S_{manualcontrol} = false$
$S_{sample}(C) = not\_collected$	

Fig. 4. First infeasible situation.

D. Unrealizability detection

The behavior specification described in the previous section is translated to a set of CSPs. While solving the CSPs, the situation in Fig. 4 is reported to be infeasible.

We can see that the there is no solution for the situation where the battery level of the UAV is below 5% and the UAV has not collected all the air samples yet. In this case, the rule requiring the UAV to land conflicts with the rule requiring the UAV to collect the air samples. We also realize that the same will happen when the battery level of the UAV is between 5% and 20%. The question to the developer is: What should the UAV do in these situations? Since in this mission the safety rules are more important than the goal to collect air samples, we relax the rule in Eq. (20) by replacing it with the following rule.

$$(S_{path}(x) = exist) \wedge (S_{battery} = above\_20\%) \Rightarrow S^G_{sample}(x) = collected \quad (23)$$

However, after updating the rule, the above situation is still reported to be infeasible. It is because the rule of being home at the end of the mission conflicts with the rule stating that the UAV must land immediately if the battery level is below 5%. If the battery level is below 5% and the UAV is not at home yet, there is no solution that can satisfy both rules. In this case, we want the UAV to land instead of still trying to go home. We therefore replace the rule in Eq. (21) by the rule below. The new rule states that the UAV should only try to be at home if its battery level is more than or equal to 5%.

$$\neg(S^I_{battery} = below\_5\%) \Rightarrow S^G_{location} = home \quad (24)$$

The situation in Fig. 4 is now feasible. However, the specification is still not realizable yet. Another situation reported to be infeasible is shown in Fig. 5.

From this situation, we realize that we did not specify what the UAV should do if there exists no path to go home. The first situation where the UAV needs to go home is after it collected air samples at all checkpoints. If there is no path, we allow the UAV not to go home anymore, which means that the UAV can land at its current location. Therefore, we replace the rule in Eq. (24) by the rule below.

$$(S_{path}(home) = exist) \wedge \neg(S^I_{battery} = below\_5\%) \Rightarrow S^G_{location} = home \quad (25)$$

Note that, depending on the missions, one may define extra actions to deal with such situations. In this paper, for the ease of presentation, we simply allow the UAV to land at its current position.

The second situation where the UAV needs to go home is when its battery level is between 5% and 20%. We again allow the UAV to land at its current location if there exists no path to go home. Thus, we replace the rule in Eq. (18) by the following rules which explicitly expresses what the UAV must do if there exists a path to go home and if there exists no path to go home.

$$(S_{battery} = from\_5\%\_to\_20\%) \wedge (S\_flying = flying) \wedge \neg(S\_location = home) \wedge (S_{path}(home) = exist) \Rightarrow \text{Exec}(\mathbf{Navigate}(home)) \quad (26)$$

$$(S_{battery} = from\_5\%\_to\_20\%) \wedge (S\_flying = flying) \wedge \neg(S\_location = home) \wedge (S_{path}(home) = not\_exist) \Rightarrow \text{Exec}(\mathbf{Land}) \quad (27)$$

The last conflict being detected is about the manual control mode. Thanks to the reported situation in Fig. 6, we realize that the rule about executing the **ManuallyFly** action in Eq. (17) conflicts with the rules requiring the UAV to return to home or land when the battery level is low. For example, if the UAV is requested to switch to the manual control mode while its battery level is below 5%, the UAV does not know what to do. Since in our scenario we prefer the UAV to land, we refine the rule about the manual control mode by only allowing the UAV to execute the **ManuallyFly** action if the battery level is above 20% as in Eq. (28).

$$(S_{manualcontrol} = true) \wedge (S_{flying} = flying) \wedge (S_{battery} = above\_20\%) \Rightarrow \text{Exec}(\mathbf{ManuallyFly}) \quad (28)$$

The examples above show the capability of our approach to detect the unrealizability of a behavior specification. This is especially useful since it is not trivial for humans to think about all situations while designing the UAV's behavior.

$S_{flying} = flying$	$S_{path}(A) = exist$
$S_{location} = other$	$S_{path}(B) = exist$
$S_{battery} = from\_5\_to\_20\%$	$S_{path}(C) = exist$
$S_{sample}(A) = collected$	$S_{path}(home) = not\_exist$
$S_{sample}(B) = collected$	$S_{manualcontrol} = false$
$S_{sample}(C) = collected$	

Fig. 5. Second infeasible situation.

$S_{flying} = flying$	$S_{path}(A) = exist$
$S_{location} = other$	$S_{path}(B) = exist$
$S_{battery} = from\_5\_to\_20\%$	$S_{path}(C) = exist$
$S_{sample}(A) = collected$	$S_{path}(home) = exist$
$S_{sample}(B) = collected$	$S_{manualcontrol} = true$
$S_{sample}(C) = collected$	

Fig. 6. Third infeasible situation.

Using our approach, one could gradually refine the behavior specification until it becomes realizable.

E. Policy generation

There are 7680 possible values for the state vector in the air quality monitoring mission, corresponding to 7680 CSPs. We used Choco [17], an open source constraint programming solver, to solve the CSPs. It took 94.7 seconds in total to solve all the CSPs and generate the execution policy on an Ubuntu computer with Intel i7- 7700K 4.20GHz processors.

F. Validation

The generated execution policy is validated on a DJI Matrice 100 UAV[2]. The execution policy is run on the DJI Manifold on-board computer[3] with a NVIDIA Tegra K1 processor. A video of the flight test is included in the submission[4]. Three scenarios are demonstrated in the flight test. The first scenario shows a normal flight where the UAV took off, visited three checkpoints and collected samples, then went home and landed. In the second scenario, while the UAV was navigating to the third checkpoint, a no-fly zone that covers the third checkpoint was added. The UAV then realized that there is no path to the last checkpoint and decided to go home. While the UAV was navigating to home, the human pilot requested for the manual control mode. The UAV therefore activated the **ManuallyFly** action. The human pilot kept controlling the UAV. When the battery level was below 20%, the UAV decided to terminate the **ManuallyFly** action, then went home and landed. The third scenario illustrates a situation where the battery level of the UAV decreases faster than normal. While the UAV was collecting samples at the second checkpoint, the battery level dropped below 20%. The UAV decided to stop collecting samples and go home. However, when the UAV was halfway to home, the battery level became less than 5%. Thus, it decided to land immediately.

VI. Conclusions

In this paper we present an approach to specify and generate reactive UAV behavior. The behavior is specified formally as a set of declarative logical rules. An execution policy is generated from the behavior specification by creating and solving a set of CSPs (Constraint Satisfaction Problems). Using constraint programming, reaction rules and goals can be easily taken into account by being translated to constraints of the CSPs. The CSPs can be solved efficiently using an existing open source solver. Any unrealizability in the behavior specification can be detected while solving the CSPs. This way, it is possible to guarantee that the generated policy is sound and complete by construction.

While the potential of our behavior specification and policy generation approach has been demonstrated and validated on a software deployed on a real UAV, a number of problems remain open. The current approach only focuses on specifying and generating feasible behavior (execution policy) without taking optimality into account. Future studies will concentrate on integrating domain specific planning algorithms into the behavior specification and generation (e.g., in the air quality monitoring scenario in Section V, one may want to use a Traveling Salesman solver to decide the visiting order of checkpoints at runtime). Our future studies will also aim at developing software toolboxes and domain specific languages to ease the usage of our behavior specification and generation approach. We also plan to apply our approach to other robotic application domains.

[2]https://www.dji.com/matrice100

[3]https://store.dji.com/product/manifold

[4]https://youtu.be/S146DP1th0U

Acknowledgment

This research is partially funded by the Research Fund KU Leuven, and by the imec-ICON project SafeDroneWare.

References

[1] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the pr2," in *International Conference on Robotics and Automation*. IEEE, 2011, pp. 5568–5575.

[2] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. C. Kemp, "Ros commander (rosco): Behavior creation for home robots," in *International Conference on Robotics and Automation*. IEEE, 2013, pp. 467–474.

[3] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[4] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *International Conference on Robotics and Automation*. IEEE, 2014, pp. 5420–5427.

[5] M. Colledanchise and P. Ögren, "How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees," in *International Conference on Intelligent Robots and Systems*. IEEE, 2016, pp. 424–429.

[6] P. Doherty, F. Heintz, and J. Kvarnström, "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation," *Unmanned Systems*, vol. 1, pp. 75–119, 2013.

[7] P. Doherty and J. Kvarnström, "Temporal action logics," *Foundations of Artificial Intelligence*, vol. 3, pp. 709–757, 2008.

[8] P. Doherty and J. Kvarnstram, "TALplanner: A temporal logic-based planner," *AI Magazine*, vol. 22, p. 95, 2001.

[9] J. Kvarnström, "Planning for Loosely Coupled Agents Using Partial Order Forward-Chaining." in *International Conference on Automated Planning and Scheduling*, 2011, pp. 138–145.

[10] P. Doherty, J. Kvarnstrom, M. Wzorek, F. Heintz, and G. Conte, "HDRC3: A Distributed HybridDeliberative/Reactive Architecture for Unmanned Aircraft Systems," in *Handbook of Unmanned Aerial Vehicles*. Springer, 2014, pp. 849–952.

[11] A. E. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos, "Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners," *Artificial Intelligence*, vol. 173, pp. 619–668, 2009.

[12] E. A. Emerson and others, "Temporal and modal logic." *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, vol. 995, p. 5, 1990.

[13] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Synthesis of control protocols for autonomous systems," *Unmanned Systems*, vol. 1, pp. 21–39, 2013.

[14] S. Maniatopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit, "Reactive high-level behavior synthesis for an atlas humanoid robot," in *International Conference on Robotics and Automation*. IEEE, 2016, pp. 4192–4199.

[15] M. Colledanchise, R. M. Murray, and P. Ogren, "Synthesis of Correct-by-Construction Behavior Trees," in *International Conference on Intelligent Robots and Systems*, 2017.

[16] R. Barták and D. Toropila, "Reformulating Constraint Models for Classical Planning." in *International Florida Artificial Intelligence Research Society Conference*, 2008, pp. 525–530.

[17] C. Prud'homme, J.-G. Fages, and X. Lorca, "Choco Documentation," 2016.